

# Mining malware specifications through static reachability analysis

Hugo Daniel Macedo<sup>1</sup> and Tayssir Touili<sup>1</sup>

LIAFA, CNRS and Univ. Paris Diderot, France  
{macedo,touili}@liafa.univ-paris-diderot.fr

**Abstract.** The number of malicious software (malware) is growing out of control. Syntactic signature based detection cannot cope with such growth and manual construction of malware signature databases needs to be replaced by computer learning based approaches. Currently, a single modern signature capturing the semantics of a malicious behavior can be used to replace an arbitrarily large number of old-fashioned syntactical signatures. However teaching computers to learn such behaviors is a challenge. Existing work relies on dynamic analysis to extract malicious behaviors, but such technique does not guarantee the coverage of all behaviors. To sidestep this limitation we show how to learn malware signatures using *static* reachability analysis. The idea is to model binary programs using pushdown systems (that can be used to model the stack operations occurring during the binary code execution), use reachability analysis to extract behaviors in the form of trees, and use subtrees that are common among the trees extracted from a training set of malware files as signatures. To detect malware we propose to use a tree automaton to compactly store malicious behavior trees and check if any of the subtrees extracted from the file under analysis is malicious. Experimental data shows that our approach can be used to learn signatures from a training set of malware files and use them to detect a test set of malware that is 5 times the size of the training set.

## 1 Introduction

Malware (malicious software) is software developed to damage the system that executes it, e.g.: virus, trojans, rootkits, etc. A malware variant performs the same damage as another known malware, but its code, its syntactical representation, is different. Malware can be grouped into families, sets of malware sharing a common trait. Security reports acknowledge a steady increase in the number of new malware. For instance, in 2010 the number of newly unique variants of malware was 286 million [13] and recent numbers confirm the trend [21]. Such numbers challenge current malware detection technology and because variants can be automatically generated the problem tends to get worse. Research confirms the unsuitability of current malware detectors [14,24]. The problem is the low-level of the techniques used.

The basic detection technique is signature matching, it consists in the inspection of the binary code and search for patterns in the form of binary sequences [27]. Such patterns, malware signatures in the jargon and syntactic signatures throughout this paper,

are manually introduced in a database by experts. As it is possible to automatically generate an unbounded number of variants, such databases would have to grow arbitrarily, not to mention it takes about two months to manually update them [14].

An alternative to signature detection is dynamic analysis, which runs malware in a virtual machine. Therefore, it is possible to check the program behavior, for instance to detect calls to system functions or changes in sensitive files, but as the execution duration must be limited in time it is difficult to trigger the malicious behaviors, since these may be hidden behind user interaction or require delays.

To overcome the problems of the previous techniques, a precise notion of malicious behavior was introduced. Such is the outcome of the recent use of model-checking techniques to perform virus detection [3,9,11,16,17,18,26,24,25,22]. Such techniques allow to check the behavior (not the syntax) of the program without executing it. A malicious behavior is a pattern written as a logical formula that specifies at a semantic level how the syntactic instructions in the binary executable perform damage during execution. As the malicious behavior is the same in all the variants of a malware, such patterns can be used as modern (semantic) signatures which can be efficiently stored.

The prime example of a malicious behavior is self-replication [27]. A typical instance of such behavior is a program that copies its own binary representation into another file, as exemplified in the assembly fragment of Fig. 1. The attacker program discovers and stores its file path into a memory address  $m$  by calling the *GetModuleFileName* function with 0 as first parameter and  $m$  as second parameter. Later such file name is used to infect another file by calling *CopyFile* with  $m$  as first parameter. Such malicious behaviors can naturally be defined in terms of system functions calls and data flow relationships.

```

 $l_1$  : push m
 $l_2$  : mov ebx 0
 $l_3$  : push ebx
 $l_4$  : call GetModuleFileName
 $l_5$  : push m
 $l_6$  : call CopyFile

```

**Fig. 1.** Malware assembly fragment.

*System functions* are the mediators between programs and their environment (user data, network access,...), and as those functions can be given a fixed semantics, and are defined in an Application Programming Interface (API), they can be used as a common denominator between programs, i.e. if the syntactical representation of programs is different but both interact in the same way with the environment, the programs are semantically equivalent from an observer perspective.

A *data flow* expresses that a value outputted at a certain time instant of program execution by a function is used as an input by another function at a following instant. For example when a parameter is outputted by a system call and is used as an input of another. Such data flow relations allow us to characterize combined behaviors purported by the related system calls. For instance, in the example of Fig. 1 it is the data flow evidenced by the variable  $m$ , defined at the invocation of *GetModuleFileName* and used at the invocation of *CopyFile* that establishes the self-replication behavior.

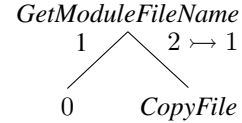
The malicious behaviors can be described naturally by trees expressing data flows among system calls made at runtime. Due to code branches during execution it is possible to have several flows departing from the same system call, thus a tree structure is particularly suitable to represent malicious behaviors. Plus, as such behaviors are described independently of the functionality of the code that makes the calls, system call

data flow based signatures are more robust against code obfuscations. Thus, a remaining challenge is to learn such trees from malware binary executables.

Recent work [2,10,14] shows that we can teach computers to learn malicious behavior specifications. Given a set of malware, the problem of extracting malicious behavior signatures consists in the extraction of the behaviors included in the set and use statistical machinery to choose the ones that are more likely to appear. However the approaches rely on dynamic analysis of executables which do not fully cover all behaviors. To overcome these limitations, in this paper we show how to use static reachability analysis to extract malicious behaviors, thus covering the whole behaviors of a program at once and within a limited time.

**Our approach.** We address such challenge in the following way: given the set of known malware binary executables, we extract its malicious behaviors in the form of edge labeled trees with two kinds of nodes. One kind represents the knowledge that a system function is called, the other kind of nodes represents which values were passed as parameters in the call (because some data flows between functions are only malicious when the calls were made with a specific parameter e.g. the 0 passed to *GetModuleFileName* in the self-replication behavior). Tree labels describe either a relation among system calls or the number of the parameter instantiated. For example, the malicious behavior displayed in Fig. 1 can be displayed in the tree shown in Fig. 2. The tree captures the self-replication behavior.

The edge on the left means that the *GetModuleFileName* function is called with 0 as first parameter (thus it will output the path to the malware file that called it) while the edge on the right captures the data flow between the two system calls i.e. the second parameter of a call to *GetModuleFileName* is an output and it is used as an input in the first parameter of a call to *CopyFile*. Thus, such tree describes the following behavior: *GetModuleFileName* is called with 0 as first parameter and its second parameter will be used as input in the first parameter of a subsequent call to *CopyFile*.



**Fig. 2.** Self-replication behavior

The first step in the tree extraction process is to model the malware binaries, which involves modeling (recursive) procedure calling and return, and parameter passing that are implemented using a stack. For this aim, we model each of the files using a push-down system (**PDS**), an automaton that mimics the binary code execution as a state transition system. With this model one is able to rigorously define the behavior of the program and use the decidable and efficient state reachability analysis of **PDSs** to calculate all the states and the contents of the stack that can occur during execution. Therefore, if malware performs a system call with certain parameters, the reachability analysis will reveal it even if the call is obfuscated, e.g.: jump to function address. The same happens if the call is made using indirect addressing because the analysis will reveal that during execution the entry point of the system call is reached. Our approach also works against bitwise manipulation of parameters, because we assume the system functions are not changed by the attacker, thus when the executions reaches the entry point of the system function, parameters must not be obfuscated, for instance in the example above even

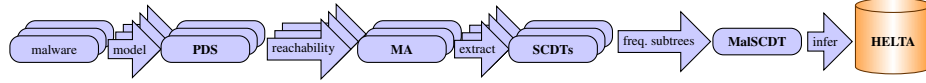
if the value of  $m$  is obfuscated, at the entry point of the call the value must be  $m$  to purport the self-replication behavior.

From the reachability analysis of each **PDS**, we obtain a multi-automaton (**MA**), a finite automaton encoding the possibly infinite reachable configurations (states and stack contents)[8,12]. As the number of system functions is finite, we cut the finite automaton to represent only the states corresponding to system function entry points and stacks limited to the finite number of parameters passed to the function.

We analyze all data flows using the **MA**s to build trees, written as system call dependency trees (**SCDTs**), representing such flows. The extracted trees correspond to a superset of the data flows present in the malware because the **PDS** model is an overapproximation of the behaviors in the binary program. This means, that when a data flow is found using our approach, there exists an execution path in the model evidencing such data flow, but such execution path may not be possible in the binary program due to approximation errors.

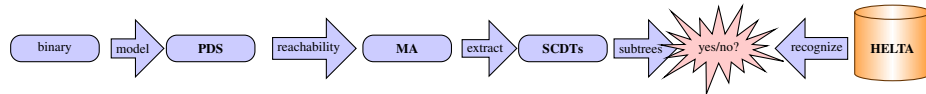
From the trees (**SCDTs**) extracted from the set of known malware binary executables we use a data-mining algorithm to compute the most frequent subtrees. We assume such correspond to malicious behaviors and we will term them malicious system call dependency trees (**MalSCDTs**). The usage of such data-mining algorithm allows to compute behaviors, which we use as signatures that are general and implementation details independent, therefore robust.

To store and recognize **MalSCDTs** we infer an automaton, termed **HELTA**, recognizing trees containing **MalSCDTs** as subtrees. This allows to efficiently store the malware signatures and recognize behaviors if they are hidden inside another behavior. The overview of the learning process from the malware files to the database of semantic signatures is depicted in Figure 3.



**Fig. 3.** Learning malicious behaviors

To evaluate the efficiency of the computed malicious behaviors, we show they can be applied to efficiently detect malware. To perform malware detection on a binary executable, we extract trees using the same procedure used in the learning process (described above), but applied to a single file. We check whether the automaton storing malicious behaviors accepts any subtree of the extracted trees (**SCDTs**). If that is the case the executable contains a malicious behavior and is classified as malware. The depiction of such process is shown in Figure 4.



**Fig. 4.** Malware detection

We implemented a tool that extracts the behaviors and selects the malicious candidates using an algorithm for the frequent subgraph problem<sup>1</sup>. With such tool we were able to infer some signatures not inferred using previous approaches [2,10,14] because our signatures track calls to functions of the Win32 API instead of calls to the Native API. It is a fact that it is always possible to use the previous approaches to find Native API level signatures equivalent to the ones we infer, therefore we do not claim our tool can express more behaviors, instead we claim that our approach is complementary to such works. It allows to express behaviors at different API levels and to extract more abstract/readable (Win32 API level) signatures.

We obtained promising results, and we were able to detect 983 malware files using the malicious trees inferred from 193 malware files, with a 0% false positive rate (thus showing our approach learns malicious behaviors that do not appear in benign programs). This number of detected malware is larger than the 16 files reported in [10] and in line with the 912 files detected in [14]. Our false positive detection rate is better (5% reported in [2]).

*Outline.* In Section 2 we show how to model binary executables as **PDSs**. Malware signatures are defined as labeled trees in Section 3. We present an algorithm to infer malware specifications in Section 4, and we show how to use tree automata to perform malware detection in Section 5. Experimental data shows our approach can be used to detect malware as detailed in Section 6. The related work is summarized in Section 7 and in Section 8 we present conclusions and future work.

## 2 Binary code modeling

Malware detection is performed directly in the executable encoding of the software (binary code containing machine instructions and data). By modeling the operational semantics of binary code, we are able to analyze it without relying on execution. This section introduces the modeling framework and how we model executable files.

### 2.1 Pushdown systems

A pushdown system (**PDS**) is a triple  $\mathcal{P} = (P, \Gamma, \Delta)$  where  $P$  is a finite set of control points,  $\Gamma$  is a finite alphabet of stack symbols, and  $\Delta \subseteq (P \times \Gamma) \times (P \times \Gamma^*)$  is a finite set of transition rules. A configuration  $\langle p, \omega \rangle$  of  $\mathcal{P}$  is an element of  $P \times \Gamma^*$ . We write  $\langle p, \gamma \rangle \hookrightarrow \langle q, \omega \rangle$  instead of  $((p, \gamma), (q, \omega)) \in \Delta$ . The immediate successor relation  $\rightsquigarrow_{\mathcal{P}} \subseteq (P \times \Gamma^*) \times (P \times \Gamma^*)$  is defined as follows: if  $\langle p, \gamma \rangle \hookrightarrow \langle q, \omega \rangle$ , then  $\langle p, \gamma\omega' \rangle \rightsquigarrow_{\mathcal{P}} \langle q, \omega\omega' \rangle$  for every  $\omega' \in \Gamma^*$ . The reachability relation  $\Rightarrow$  is defined as the reflexive and transitive closure of the immediate successor relation.

Given a set of configurations  $C$ ,  $post(C)$  is defined as the set of immediate successors of the elements in  $C$ . The reflexive and transitive closure of  $post$  is denoted as  $post^*(C) = \{c' \in P \times \Gamma^* \mid \exists c \in C, c \Rightarrow c'\}$ . Analogously  $pre(C)$  is defined as the set of immediate predecessors of elements in  $C$ . Its reflexive and transitive closure is denoted as  $pre^*(C) = \{c \in P \times \Gamma^* \mid \exists c' \in C, c \Rightarrow c'\}$ .

<sup>1</sup> A tree is a special case of a graph

Given a pushdown system  $\mathcal{P} = (P, \Gamma, \Delta)$ , a  $\mathcal{P}$ -multi-automaton,  $\mathcal{P} - MA$  or **MA** when  $\mathcal{P}$  is clear from context, is a tuple  $\mathcal{A} = (\Gamma, Q, \delta, P, F)$ , where  $Q$  is a finite set of states,  $\delta \subseteq Q \times \Gamma \times Q$  is a transition relation,  $P \subseteq Q$  is the set of initial states corresponding to the control points of  $\mathcal{P}$ , and  $F \subseteq Q$  is a set of final states.

The transition relation for **MA** is the smallest relation  $\rightarrow \subseteq Q \times \Gamma^* \times Q$  satisfying:

- $q \xrightarrow{\gamma} q'$  if  $(q, \gamma, q') \in \delta$
- $q \xrightarrow{\omega\gamma} q'$  if  $q \xrightarrow{\omega} q''$  and  $q'' \xrightarrow{\gamma} q'$

$\mathcal{A}$  accepts (recognizes) a configuration  $\langle p, w \rangle$  if  $p \xrightarrow{w} q$  for some  $q \in F$ . The set of configurations recognized by a **MA**  $\mathcal{A}$  is called regular and is designated by  $Conf(\mathcal{A})$ . The  $post^*$  and  $pre^*$  of regular configurations can be efficiently computed:

**Theorem 1.** [8,12] *For a pushdown system  $\mathcal{P} = (P, \Gamma, \Delta)$  and **MA**  $\mathcal{A}$ , there exist **MA**s  $\mathcal{A}_{post^*}$  and  $\mathcal{A}_{pre^*}$  recognizing  $post^*(Conf(\mathcal{A}))$  and  $pre^*(Conf(\mathcal{A}))$  respectively. These can be constructed in polynomial time and space.*

## 2.2 Modeling binary programs with PDSs

We use the approach detailed in [24, Section 2] to model each executable program  $\mathbb{P}$ . The approach relies on the assumption that there exists an oracle  $\mathcal{O}$  computing a **PDS**  $\mathcal{P} = (P, \Gamma, \Delta)$  from the binary program, where  $P$  corresponds to the control points of the program,  $\Gamma$  corresponds to the approximate set of values pushed to the stack, and  $\Delta$  models the different instructions of the program. The obtained **PDS** mimics the runs of program  $\mathbb{P}$ .

In addition to the approach of [24], let **API** be the set of all Application Programming Interface function names available in the program. We assume the oracle  $\mathcal{O}$  approximates the set  $P_{API} \subseteq P$  of control points of a program that correspond to instruction addresses that at program runtime are translated (dynamically linked) by the operating system into system function entry points, the number of parameters of such functions and the type of each parameter. We consider a simple type system:  $\tau ::= in \mid out$  (*in* for input parameter, and *out* for output) containing the atomic value *out* used to denote a parameter that is modified after function execution and *in* to denote the parameter is not changed by the function.

We assume,  $\mathcal{O}$  computes a function  $\varrho_\lambda : P_{API} \rightarrow \mathbf{API}$  that identifies program control points corresponding to system calls with an unique function name, a function  $\varrho_\tau : P_{API} \times \mathbb{N} \rightarrow 2^\tau$  such that  $\varrho_\tau(p, n)$  is the set<sup>2</sup> of possible types of the  $n$ -th parameter of the system call that has  $p$  as entry point, and a function  $\varrho_{ar} : P_{API} \rightarrow \mathbb{N}$  defining the number of parameters for each system call in  $P_{API}$ . For example, if we consider the program of Fig. 1, we obtain  $P_{API} = \{l_g, l_c\}$  since these two points correspond to system call entry points,  $\varrho_\lambda(l_g) = GetModuleFileName$  since  $l_g$  corresponds to the entry point of the function *GetModuleFileName*.  $\varrho_{ar}(l_g) = 3$  since *GetModuleFileName* has three parameters, and  $\varrho_\tau(l_g, 2) = \{out\}$  since the second parameter of the *GetModuleFileName* function is defined as an output, and analogously  $\varrho_\tau(l_g, 1) = \varrho_\tau(l_g, 3) = \{in\}$ , since these correspond to input parameters.

<sup>2</sup> The API defines parameters that are both input and output.

### 3 Malicious behavior specifications

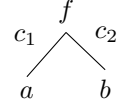
As already mentioned, malicious behaviors, data flow relationships between system function calls, will be expressed as trees where nodes represent system functions or parameter values and edges specify the data flow or the number of the parameter to which the value was passed. We will now formally introduce the notion of edge labeled trees.

#### 3.1 Edge labeled trees

An unranked alphabet is a finite set  $\mathcal{F}$  of symbols. Given an unranked alphabet  $\mathcal{F}$ , let a set of colors  $\mathcal{C}$  be an alphabet of unary symbols and disjoint from  $\mathcal{F}$ , and  $\mathcal{X}$  be a set of variables disjoint from  $\mathcal{F}$ . The set  $\mathcal{T}(\mathcal{F}, \mathcal{C}, \mathcal{X})$  of colored terms over the unranked alphabet  $\mathcal{F}$ , colors  $\mathcal{C}$  and variables  $\mathcal{X}$  is the smallest set of terms such that:

- $\mathcal{F} \subseteq \mathcal{T}(\mathcal{F}, \mathcal{C}, \mathcal{X})$ ,
- $\mathcal{X} \subseteq \mathcal{T}(\mathcal{F}, \mathcal{C}, \mathcal{X})$ , and
- $f(c_1(t_1), \dots, c_n(t_n)) \in \mathcal{T}(\mathcal{F}, \mathcal{C}, \mathcal{X})$ , for  $n \geq 1$ ,  $c_i \in \mathcal{C}$ ,  $t_i \in \mathcal{T}(\mathcal{F}, \mathcal{C}, \mathcal{X})$ .

If  $\mathcal{X} = \emptyset$  then  $\mathcal{T}(\mathcal{F}, \mathcal{C}, \mathcal{X})$  is written as  $\mathcal{T}(\mathcal{F}, \mathcal{C})$ , and its elements are designated as ground terms. Each element of the set of terms can be represented by an edge labeled tree. For example, let  $\mathcal{F} = \{f\}$ ,  $\mathcal{C} = \{c_1, c_2\}$ , and  $\mathcal{X} = \emptyset$ . The colored tree  $f(c_1(a), c_2(b)) \in \mathcal{T}(\mathcal{F}, \mathcal{C})$  can be represented by the edge labeled tree of Fig. 5.



**Fig. 5.** Example

Let  $\mathcal{X}_n$  be a set of  $n$  variables. A term  $E \in \mathcal{T}(\mathcal{F}, \mathcal{C}, \mathcal{X}_n)$  is called an environment and the expression  $E[t_1, \dots, t_n]$  for  $t_1, \dots, t_n \in \mathcal{T}(\mathcal{F}, \mathcal{C})$  denotes the term in  $\mathcal{T}(\mathcal{F}, \mathcal{C})$  obtained from  $E$  by replacing the variable  $x_i$  by  $t_i$  for each  $1 \leq i \leq n$ .

A subtree  $t'$  of a tree  $t$  in  $\mathcal{T}(\mathcal{L}, \mathcal{C})$ , written as  $t' \triangleleft t$ , is a term such that there exists an environment  $E$  in  $\mathcal{T}(\mathcal{L}, \mathcal{C}, \{x\})$  where  $x$  appears only once and  $t = E[t']$ .

The tree  $f(c_1(a), c_2(b))$  represents the same behavior as tree  $f(c_2(b), c_1(a))$ . Thus, to efficiently compare edge labeled trees, and to avoid missing malicious behaviors due to tree representation, we define a canonical representation of edge labeled trees. We assume that  $\mathcal{F}$  and  $\mathcal{C}$  are totally ordered.

A term is in canonical form if it is a constant (leaf) or if it is a function (tree node) where each argument is in canonical form and arguments are sorted without repetitions by term order.

Let  $c \in \mathcal{C}$  and  $t \in \mathcal{F}(\mathcal{C}, \mathcal{T})$  such that  $\mathcal{F}$ ,  $\mathcal{C}$ , and  $\mathcal{T}$  are respectively ordered by  $<^{\mathcal{F}}$ ,  $<^{\mathcal{C}}$ , and  $<^{\mathcal{T}}$ , and  $t$  is in canonical form. We assume a subtree insertion operation (**insert\_subtree**) where **insert\_subtree**( $c(t)$ ,  $t'$ ) adds  $c(t)$  as a child to the root of  $t'$  in the correct place to maintain a canonical representation of the tree, overwriting if the subtree  $c(t)$  already exists.

#### 3.2 System call dependency trees

We will represent malware behaviors as trees encoding data flow relationships between system function calls. Tree nodes represent either system functions or parameter values.



Edge colors label the characteristics of the data flow between functions, e.g.  $2 \mapsto 1$  labeling an edge from function  $f$  and  $f'$  means that at some point  $f$  is called with some value  $v$  as second parameter, which is of type *out*, and afterwards  $f'$  is called with  $v$  as first parameter, which in turn is of type *in*. Moreover, when an edge connects a node labeled with function  $f$  and a child node with some value  $v$ , meaning the function was called with parameter  $v$ , it will be labeled with the number of the parameter, thus to represent a call was made with 0 as first parameter to function  $f$ , we add 1 as a label of the edge from node  $f$  to node 0.

**Definition 1.** Formally, let  $\mathcal{F}$  be the set of all system call function names (the union of all possibly **API** function names returned by the oracle of Section 2) and values passed as function parameters (a subset of the union of all  $\Gamma$  sets calculated by the oracle). In addition, let  $\mathcal{C}$  be a set of colors containing all the possible parameter numbers and data flows, i.e.:  $\mathcal{C} = \{1, \dots, \max_{f \in \mathbf{API}}(\varrho_{ar}(f))\} \cup \{x \mapsto y \mid x, y \in \{1, \dots, \max_{f \in \mathbf{API}}(\varrho_{ar}(f))\}\}$  A System Call Dependency Tree, written as **SCDT**, is defined as a ground term of the set  $\mathcal{T}(\mathcal{F}, \mathcal{C})$ .

**Example.** Let  $\mathcal{F} = \{0, \text{GetModuleFileName}, \text{CopyFile}\}$  and  $\mathcal{C} = \{1, 2 \mapsto 1\}$ , the behavior of Fig. 2 can be described by  $t = \text{GetModuleFileName}(1(0), 2 \mapsto 1(\text{CopyFile}))$ .

## 4 Mining malware specifications

In this section we show how to compute the **SCDTs** corresponding to malware behaviors that we will use as malware specifications. Given a finite set of programs  $\mathbb{P}_1, \dots, \mathbb{P}_q$  known to be malicious in advance we compute **PDSs**  $\mathcal{P}_1, \dots, \mathcal{P}_q$  that model these malicious programs. Then, for each **PDS**  $\mathcal{P}_i$  we compute a set of trees  $\mathbb{TS}_i$  that contains the data flows represented as **SCDTs** for the program  $\mathbb{P}_i$ . From the computed set of trees for each program,  $\mathbb{TS}_1, \dots, \mathbb{TS}_q$ , we calculate the common subtrees, the ones that are most probable to appear in malware, that we use as malware specifications.

To compute the sets of trees  $\mathbb{TS}_i$  we proceed as follows: For each program  $\mathbb{P}_i$  modeled as a **PDS**  $\mathcal{P}_i$  we compute the finite automaton encoding the set of reachable configurations from the initial state using the reachability analysis algorithm from [12]. As there may be an infinite number of configurations and we are only interested in the configurations whose control points correspond to a system function entry point with some finite number of elements in the stack (only the parameters of the function under consideration are important), we build another automaton recognizing such finite set of configurations. For each of such configurations, understood as possible data flow origins, we repeat the process to calculate the reachable configurations, understood as possible data flow destinations. Then, if a data flow between configurations is found, i.e. the value passed as a parameter to an origin configuration has type *out* and the same value passed as a parameter of type *in* to a destination configuration, we build a **SCDT** with the origin function as root node and an edge to a node corresponding to the destination function.

To calculate the common subtrees we use the algorithm [30] computing frequent subgraph, to compute frequent subtrees.



#### 4.1 System call targeted reachability analysis

To compute the data flows for a malware pushdown system model  $\mathcal{P} = (P, \Gamma, \Delta)$ , we first calculate the reachability of  $\mathcal{P}$  using the algorithms presented in [12]. From  $\mathcal{P}$  we build the **(MA)** automaton  $\mathcal{A}$  that recognizes the  $post^*(\langle p_i, \epsilon \rangle)$ , i.e. the set of reachable configurations from the initial configuration  $\langle p_i, \epsilon \rangle$ , where  $p_i$  is a designated initial control point and  $\epsilon$  denotes the empty stack.

**MA Trimming.** To compute data flows between system call related control points  $p_o, p_d \in P_{API}$  with parameter numbers  $\varrho_{ar}(p_o) = m$  and  $\varrho_{ar}(p_d) = n$  we need to consider only the top  $m + 1$  and  $n + 1$  elements of the stack reached at control points  $p_o$  and  $p_d$  because, in assembly, parameters are passed to functions through the stack. Before invoking a function the parameters are pushed in reverse order into the stack, and after the return address is pushed. Thus, if a function receives  $m$  parameters, then at its entry point, for instance  $p_o$ , the top  $m + 1$  elements of the stack correspond to the parameters plus the return address. Thus we only need to consider the top  $m + 1$  elements of the stack reached at control point  $p_o$ . This is the reason why we can analyze the possibly infinite number of configurations encoded in the reachability resulting finite automaton, we only inspect a finite subset. To abbreviate the algorithm that computes **SCDT** we define such subset of configurations in terms of a new automaton obtained by cutting the **MA** resulting from the reachability analysis.

**Definition 2.** Given a **MA**  $\mathcal{A}$  recognizing the reachable configurations of a **PDS**  $\mathcal{P} = (P, \Gamma, \Delta)$  we define the trim automaton  $\mathcal{A}^\dagger$  as the automaton recognizing the configurations in the set:  $\{\langle p, w \rangle \in P_{API} \times \Gamma^* \mid |w| = \varrho_{ar}(p) + 1 \wedge \exists w' \in \Gamma^* \text{ s.t. } \langle p, ww' \rangle \text{ is accepted by } \mathcal{A}\}$

Intuitively, we cut the automaton and keep only configurations where control points  $p$  correspond to system function entry points, and the stacks are bounded by the number of parameters of the function plus one to take into account the return address. The trim operation will be written as  $\Psi$ , thus  $\mathcal{A}^\dagger = \Psi(\mathcal{A})$ . It is trivial to prove that the  $Conf(\mathcal{A}^\dagger)$  is a finite language, in fact the number of configurations corresponding to valid system call function entry point, and its finite number of parameters is at most:  $O(|P_{API}| \cdot |\Gamma| \cdot \max_{p \in P_{API}}(\varrho_{ar}(p)))$ .

#### 4.2 Extracting SCDTs

Algorithms 1 and 2 detail our approach to extract behaviors. We assume a maximum tree height  $h \in \mathbb{N}$  is given as input. We write  $\omega[n]$  to denote the  $n$ -th element of some word  $\omega \in \Gamma^*$ .

The Algorithm 1 iterates over the models  $\mathcal{P}_1, \dots, \mathcal{P}_q$  (line 1). For each it initializes the set of resulting trees to the empty set (line 2) and computes the configurations corresponding to system calls that are reachable from the given initial configuration  $\langle p_i, \epsilon \rangle$  (line 3). The initial configuration is built using the binary executable entry point

---

##### Algorithm 1: ExtractSCDT

---

```

1 forall the  $\mathcal{P}_i$  do
2    $TS_i \leftarrow \emptyset$ ;
3    $\mathcal{A}_i^\dagger \leftarrow \Psi(post^*(\langle p_i, \epsilon \rangle))$ ;
4   forall the  $\langle p_o, \omega_o \rangle \in Conf(\mathcal{A}_i^\dagger)$  do
5      $TS_i \leftarrow TS_i \cup \{BuildSCDT(\langle p_o, \omega_o \rangle, h)\}$ ;
6   end
7 end
8 return  $TS$ ;
```

---

and an empty stack. Then, for every configuration corresponding to a system call entry point  $\langle p_o, \omega_o \rangle$  recognized by the trim automaton (line 4) it calls **BuildSCDT** to build a **SCDT** tree of height at most  $h$  with the function of entry point  $p_o$  as root (line 5).

The **BuildSCDT** procedure is displayed in Algorithm 2, it is used to recursively build a tree. First, the tree to be returned is initialized to be the origin system call entry point  $p_o$  (line 1). When the maximum desired tree height is not reached (line 2), we calculate what are the system calls reached from  $\langle p_o, \omega_o \rangle$  (line 3) and check for flows to any system call related configuration  $\langle p_d, \omega_d \rangle$  (line 4). If a data flow is found between two configurations (line 5), i.e. there are parameter numbers  $n$  and  $m$  such that the value passed to system call at control point  $p_o$  is the same as the value passed in position  $m$  of system call at a control point  $p_d$ , and there is in fact a flow (line 6) i.e. the parameter  $n$  of the function corresponding to the entry point  $p_o$  is of type *out* and the parameter  $m$  of the function corresponding to the entry point  $p_d$  is of type *in*, we add a new child with label  $n \mapsto m$  to the recursively computed tree for the destination system call  $p_d$  (line 7).

---

**Algorithm 2: BuildSCDT**


---

```

1 tree =  $\varrho_\lambda(p_o)$ ;
2 if  $h > 0$  then
3    $\mathcal{A}^\dagger \leftarrow \Psi(\text{post}^*(\langle p_o, \omega_o \rangle))$ ;
4   forall the  $\langle p_d, \omega_d \rangle \in \text{Conf}(\mathcal{A}^\dagger) \setminus \{\langle p_o, \omega_o \rangle\}$  do
5     forall the  $(n, m)$  s.t.  $1 \leq n \leq \varrho_{ar}(p_o) \wedge 1 \leq m \leq \varrho_{ar}(p_d)$  do
6       if  $w_o[n] = w_d[m] \wedge \varrho_\tau(p_o, n) = \text{out} \wedge \varrho_\tau(p_d, m) = \text{in}$  then
7         tree  $\leftarrow \text{insert\_subtree}(n \mapsto m(\text{BuildSCDT}(\langle p_d, \omega_d \rangle, h - 1)), \text{tree})$ ;
8       end
9     end
10  end
11 end
12 forall the  $n \in \{1, \dots, \varrho_{ar}(p_o)\}$  do
13   tree  $\leftarrow \text{insert\_subtree}(n(w_o[n]), \text{tree})$ ;
14 end
15 return tree;
```

---

To add the edges representing the values passed as parameters in the call of  $p_o$  we iterate over the possible number of parameters of the origin system call entry point (line 12) and add an edge with the number of parameter  $n$  and the value passed in the stack  $w_o[n]$  (line 13). When the maximum desired tree height is reached, the algorithm returns only a tree with  $p_o$  as root and the values passed as parameters in the call.

### 4.3 Computing malicious behavior trees

After extracting **SCDTs** for each of the inputted malware programs, one has to compute which are the ones that correspond to malicious behaviors. The **SCDTs** that correspond to malicious behaviors will be named malicious trees. To choose the malicious trees we compute the most frequent subtrees in the set  $\mathbb{T}\mathbb{S}$  of trees extracted from the set of malware used to train our detector. For that we need the notion of *support set*, the set of trees containing some given subtree, and the notion of *tree support* that gives the ratio of trees containing the subtree to the whole set of trees.

Given a finite set of trees  $\mathbb{T}\mathbb{S} \subseteq \mathcal{T}(\mathcal{F}, \mathcal{C})$  and a tree  $t \in \mathbb{T}\mathbb{S}$ , the *support set* of a tree  $t$  is defined as  $T_t = \{t' \mid t \triangleleft t', t' \in \mathbb{T}\mathbb{S}\}$ . The *tree support* of a tree  $t$  in the set  $\mathbb{T}\mathbb{S}$  is

calculated as  $\text{sup}(t) = \frac{|T_t|}{|\mathcal{TS}|}$ . For a fixed threshold  $k$  the set of frequent trees of  $T$  is the set of trees with *tree support* greater than  $k$ .

**Definition 3.** For a set of system call dependency trees  $\mathcal{TS} \subseteq \mathcal{T}(\mathcal{F}, \mathcal{C})$  and a given threshold  $k$ , a malicious behavior tree is a tree  $t \in \mathcal{TS}$  s.t.  $\text{sup}(t) \geq k$ . The set of malicious behavior trees will be called **MalSCDT**.

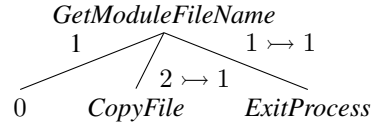
To compute frequent subtrees we specialize the frequent subgraph algorithm presented in [30] to the case of trees. The algorithm receives a set of trees and a support value  $k \in [0, 1]$  and outputs all the subtrees with support at least  $k$ . The graph algorithm works by defining a lexicographical order among the trees and mapping each to a canonical representation using a code based on the depth-first search tree generated by the traversal. Using such lexicographical order the subtree search space can be efficiently explored avoiding duplicate computations.

## 5 Malware detection

We show in this section how the malicious behaviors trees that we computed using our techniques can be used to efficiently detect malware. To decide whether a given program  $\mathbb{P}$  is malware or not, we apply again the technique described in Section 4 to compute the **SCDTs** for the program  $\mathbb{P}$  being analyzed. Then we check whether such trees correspond to malicious behaviors, i.e. whether such trees contain subtrees that correspond to malicious behaviors.

To efficiently perform this task, we use tree automata. The advantage of using tree automata is that we can build the minimal automaton that recognizes the set of malicious signatures, to obtain a compact and efficient database. Plus, malware detection, using membership in automata, can be done efficiently. However, we need to adapt tree automata to suite malware detection, that is, to define automata that can recognize *edge labeled* trees. Furthermore, we cannot use standard tree automata because the trees that can be generated from the program  $\mathbb{P}$  to be analyzed may have arbitrary arities (since we do not know a priori the behaviors of  $\mathbb{P}$ ). For example the behavior of the program  $\mathbb{P}$  can be described by the tree of Fig. 6 that contains the self-replication malicious behavior of Fig. 2. However, if we use a binary tree automaton  $\mathbb{H}$  to recognize the tree of Fig. 2,  $\mathbb{H}$  will not recognize the tree of Fig. 6, because  $\mathbb{P}$  contains the malicious behaviors and extra behaviors. To overcome this problem we will use unranked tree automata (a.k.a. hedge automata), since the trees that can be obtained by analysing program  $\mathbb{P}$  might have arbitrary arity.

In this section, we show how to use hedge automata for malware detection. First, we give the formal definition of hedge automata. Then, we show how we can infer a hedge automaton to recognize malicious behaviors that may be contained in some tree. And we conclude by explaining how to use it to detect malware.



**Fig. 6.** Behaviors extracted from  $\mathbb{P}$

### 5.1 Tree automata for edge labeled trees

**Definition 4.** An hedge edge labeled tree automaton (**HELTA**) over  $\mathcal{T}(\mathcal{F}, \mathcal{C})$  is a tuple  $\mathbb{H} = (Q^{\mathbb{H}}, \mathcal{F}, \mathcal{C}, \mathcal{A}, \Delta^{\mathbb{H}})$  where  $Q^{\mathbb{H}}$  is a finite set of states,  $\mathcal{A} \subseteq Q^{\mathbb{H}}$  is the set of final states, and  $\Delta^{\mathbb{H}}$  is a finite set of rewriting rules defined as  $f(R) \rightarrow q$  for  $f \in \mathcal{F}$ ,  $q \in Q^{\mathbb{H}}$ , and  $R \subseteq [\mathcal{C}(Q^{\mathbb{H}})]^*$  is a regular word language over  $\mathcal{C}(Q^{\mathbb{H}})$  i.e. the language encoding all the possible children of the tree node  $f$ .

We define a move relation  $\rightarrow_{\mathbb{H}}$  between ground terms in  $\mathcal{T}(\mathcal{F} \cup Q^{\mathbb{H}}, \mathcal{C})$  as follows: Let  $t, t' \in \mathcal{T}(\mathcal{F} \cup Q^{\mathbb{H}}, \mathcal{C})$ , the move relation  $\rightarrow_{\mathbb{H}}$  is defined by:  $t \rightarrow_{\mathbb{H}} t'$  iff there exists an environment  $E \in \mathcal{T}(\mathcal{F} \cup Q^{\mathbb{H}}, \mathcal{C}, \{x\})$ , a rule  $r = f(R) \rightarrow q \in \Delta^{\mathbb{H}}$  such that  $t = E[f(c_1(q_1), \dots, c_n(q_n))]$ , and  $c_1(q_1) \dots c_n(q_n) \in R$ , and  $t' = E[q]$ . We write  $\xrightarrow{*}_{\mathbb{H}}$  to denote the reflexive and transitive closure of  $\rightarrow_{\mathbb{H}}$ . Given an **HELTA**  $\mathbb{H} = (Q^{\mathbb{H}}, \mathcal{F}, \mathcal{C}, \mathcal{A}, \Delta^{\mathbb{H}})$  and an edge labeled tree  $t$ , we say that  $t$  is accepted by a state  $q$  if  $t \xrightarrow{*}_{\mathbb{H}} q$ ,  $t$  is accepted by  $\mathbb{H}$  if  $\exists q \in \mathcal{A}$  s.t.  $t \xrightarrow{*}_{\mathbb{H}} q$ .

Intuitively, given an input term  $t$ , a run of  $\mathbb{H}$  on  $t$  according to the move relation  $\rightarrow_{\mathbb{H}}$  can be done in a bottom-up manner as follows: first, we assign nondeterministically a state  $q$  to each leaf labeled with symbol  $f$  if there is in  $\Delta^{\mathbb{H}}$  a rule of the form  $f(R) \rightarrow q$  such that  $\epsilon \in R$ . Then, for each node labeled with a symbol  $f$ , and having the terms  $c_1(t_1), \dots, c_n(t_n)$  as children, we must collect the states  $q_1, \dots, q_n$  assigned to all its children, i.e., such that  $c_i(t_i) \xrightarrow{*}_{\mathbb{H}} q_i$ , for  $1 \leq i \leq n$ , and then associate a state  $q$  to the node itself if there exists in  $\Delta^{\mathbb{H}}$  a rule  $r = f(R) \rightarrow q$  such that  $q_1 \dots q_n \in R$ . A term  $t$  is accepted if  $\mathbb{H}$  reaches the root of  $t$  in a final state.

### 5.2 Inferring tree automata from malicious behavior trees

In this section we show how to infer an **HELTA** recognizing trees containing the inferred malicious behaviors. Thus, if  $t$  is a malicious behavior, and  $t'$  is a behavior of a program  $\mathbb{P}$  that is under analysis, such that  $t'$  contains the behavior described by  $t$ , the automaton must recognize it. As an example assume  $t \in \mathbf{MalSCDT}$  is a tree of the form  $f(c_1(a), c_2(b))$ , s.t.  $a, b \in \mathcal{F}$  and  $E \in \mathcal{T}(\mathcal{F}, \mathcal{C}, \{x\})$  is an environment, then the automaton must recognize trees  $t'$  of the form:  $E[f(c_1^1(t_1^1), \dots, c_{m_1}^1(t_{m_1}^1), c_1(a(e_1)), c_1^2(t_1^2), \dots, c_{m_2}^2(t_{m_2}^2), c_2(b(e_2)), c_1^3(t_1^3), \dots, c_{m_3}^3(t_{m_3}^3))]$  meaning the tree is embedded in other tree, i.e.  $t$  is a subtree of  $t'$  and it may have extra behaviors  $c_i^j(t_i^j)$  and also extra subtrees  $e_1, e_2 \in \mathcal{T}(\mathcal{F}, \mathcal{C})$  as child of the leafs  $a$  and  $b$ .

Let  $t \in \mathbf{MalSCDT}$ , we define the operation  $\Omega : \mathbf{MalSCDT} \rightarrow \mathcal{T}(\mathcal{F}, \mathcal{C})$  that transforms a malicious tree into the set of all system call dependency trees containing the malicious behavior  $t$ .  $\Omega$  is defined inductively as:

- (1)  $\Omega(a) = \{a(t) \mid t \in \mathcal{T}(\mathcal{F}, \mathcal{C})\}$ , if  $a \in \mathcal{F}$  is a leaf,
- (2)  $\Omega(f(c_1(t_1), \dots, c_n(t_n))) = \{f(c_1^1(t_1^1), \dots, c_{n_1}^1(t_{n_1}^1), c_1(\Omega(t_1)), c_1^2(t_1^2), \dots, c_{n_2}^2(t_{n_2}^2), \dots, c_1^n(t_1^n), \dots, c_{n_n}^n(t_{n_n}^n), c_n(\Omega(t_n)), c_1^{n+1}(t_1^{n+1}), \dots, c_{n_{n+1}}^{n+1}(t_{n_{n+1}}^{n+1})) \mid c_i^j(t_i^j) \in \mathcal{C} \text{ and } t_i^j \in \mathcal{T}(\mathcal{F}, \mathcal{C})\}$ , otherwise.

The first rule asserts that after the leaves of the malicious behavior  $t$  there may be other behaviors, while the second asserts that in the nodes of the tree  $t'$  there may be

extra behaviors, for instance the edge to *ExitProcess* in Fig. 6. Then, if  $t$  is a malicious behavior tree, we would like to compute an **HELTA** that recognizes all the trees  $t'$  s.t.  $\exists t'' \in \Omega(t)$  and  $t' = E[t'']$  for an environment  $E \in \mathcal{T}(\mathcal{F}, \mathcal{C}, \{x\})$ .

Let **MalSCDT** be a finite set of malicious trees, by definition each  $t \in \mathbf{MalSCDT}$  is a term of  $\mathcal{T}(\mathcal{F}, \mathcal{C})$ . We infer an **HELTA**  $\mathbb{H} = (Q^{\mathbb{H}}, \mathcal{F}, \mathcal{C}, \mathcal{A}, \Delta^{\mathbb{H}})$  recognizing trees containing malicious behaviors. Where  $Q^{\mathbb{H}} = \{q_t \mid t \triangleleft t' \text{ and } t' \in \mathbf{MalSCDT}\} \cup \{q_t \mid t \in \mathcal{F}\}$  i.e. contains a state for each subtree of the trees to accept, plus a state for each possible symbol of the alphabet that will be reached when a subtree with such symbol as root is not recognized. The final states are defined as the states that correspond to recognizing a malicious tree  $\mathcal{A} = \{q_t \mid t \in \mathbf{MalSCDT}\}$ . And  $\Delta^{\mathbb{H}}$  is defined by rules:

- R1 For all  $f \in \mathcal{F}$ ,  $f([\mathcal{C}(Q^{\mathbb{H}})]^*) \rightarrow q_f \in \Delta^{\mathbb{H}}$
- R2 For all  $t = f(c_1(t_1), \dots, c_n(t_n))$  such that  $t \triangleleft t'$  and  $t' \in \mathbf{MalSCDT}$ ,  $f([\mathcal{C}(Q^{\mathbb{H}})]^* c_1(q_{t_1}) [\mathcal{C}(Q^{\mathbb{H}})]^* \dots [\mathcal{C}(Q^{\mathbb{H}})]^* c_n(q_{t_n}) [\mathcal{C}(Q^{\mathbb{H}})]^*) \rightarrow q_{f(c_1(t_1), \dots, c_n(t_n))} \in \Delta^{\mathbb{H}}$
- R3 For all final state  $q_t \in \mathcal{A}$  and all  $f \in \mathcal{F}$ ,  $f([\mathcal{C}(Q^{\mathbb{H}})]^*, q_t, [\mathcal{C}(Q^{\mathbb{H}})]^*) \rightarrow q_t \in \Delta^{\mathbb{H}}$

Intuitively, for  $f \in \mathcal{F}$ , states  $q_f$  recognize all the terms whose roots are  $f$ . This is ensured by R1. In the rules  $[\mathcal{C}(Q^{\mathbb{H}})]^*$  allows to recognize terms  $t$  in (1) and  $c_i^j(t_i^j)$  in (2). For a subtree  $t_i$  of a malicious behavior  $t$  in every **MalSCDT**,  $q_{t_i}$  recognizes  $\Omega(q_{t_i})$ . This is ensured by rules R2, which guarantees that a malicious tree containing extra behaviors is recognized. R3 guarantees that a tree containing a malicious behavior as subtree is recognized, i.e. R3 ensures that if  $t$  is a malicious behavior and  $E \in \mathcal{T}(\mathcal{F}, \mathcal{C}, \{x\})$  is an environment, then  $q_t$  recognizes  $E[t]$  for every  $t'$  in  $\Omega(t)$ .

In the following we assert that if a tree  $t'$  contains a subtree  $t''$  that contains a malicious behavior  $t$ , then the inferred automaton will recognize it (even if there are extra behaviors). Proof should follow by induction.

**Theorem 2.** *Given a term  $t \in \mathbf{MalSCDT}$ , and  $t' \in \mathcal{T}(\mathcal{F}, \mathcal{C})$ . If there  $\exists t'' \in \Omega(t)$  and an environment  $E \in \mathcal{T}(\mathcal{F}, \mathcal{C}, \{x\})$  and  $t' = E[t'']$ , then  $t' \xrightarrow{*}_{\mathbb{H}} q_t$ .*

### 5.3 Malware detection

The detection phase works as follows. Given a program  $\mathbb{P}$  to analyze we build a **PDS** model  $\mathcal{P}$  using the approach described in Section 2, then we extract the set of behaviors  $\mathbb{TS}$  contained in  $\mathbb{P}$  using the approach in Section 4. Then we use the automaton  $\mathbb{H}$  to search if any of the trees in  $\mathbb{TS}$  can be matched by the automaton. If that is the case the program  $\mathbb{P}$  is deemed malware.

**Example.** Suppose the tree in Fig. 6 was extracted and the tree in Fig. 2 is the only malicious behavior in **MalSCDT**, which in turn is defined using  $\mathcal{C} = \{1, 2 \mapsto 1\}$  and  $\mathcal{F} = \{0, \text{CopyFile}, \text{ExitProcess}, \text{GetModuleFileName}\}$ . We define an automaton  $\mathbb{H}$  where the set of states is  $Q^{\mathbb{H}} = \{q_0, q_{\text{ExitProcess}}, q_{\text{CopyFile}}, q_{\text{GetModuleFileName}(1(0), 2 \mapsto 1(\text{CopyFile}))}\}$ , the accepting set is  $\mathcal{A} = \{q_{\text{GetModuleFileName}(1(0), 2 \mapsto 1(\text{CopyFile}))}\}$ , and  $\Delta^{\mathbb{H}}$  contains rules processing the leaves:  $0([\mathcal{C}(Q^{\mathbb{H}})]^*) \rightarrow_{\mathbb{H}} q_0$ ,  $\text{ExitProcess}([\mathcal{C}(Q^{\mathbb{H}})]^*) \rightarrow_{\mathbb{H}} q_{\text{ExitProcess}}$ , and  $\text{CopyFile}([\mathcal{C}(Q^{\mathbb{H}})]^*) \rightarrow_{\mathbb{H}} q_{\text{CopyFile}}$ . And a rule  $\text{GetModuleFileName}([\mathcal{C}(Q^{\mathbb{H}})]^*, 1(q_0), [\mathcal{C}(Q^{\mathbb{H}})]^*, 2 \mapsto 1(q_{\text{CopyFile}}), [\mathcal{C}(Q^{\mathbb{H}})]^*) \rightarrow_{\mathbb{H}} q_{\text{GetModuleFileName}(1(0), 2 \mapsto 1(\text{CopyFile}))}$  processing the whole malicious behavior of Fig. 2.

## 6 Experiments

To evaluate our approach, we implemented a tool prototype that was tested on a dataset of real malware and benign programs. The input dataset of malware contains 1176 malware instances (Virus, Backdoors, Trojans, Worms,...) collected from virus repositories as VX Heavens and a disjoint dataset of 250 benign files collected from a Windows XP fresh operating system installation. We arbitrarily split the malware dataset into a training and test group. The train dataset was used to infer the malicious trees that were used in the detection of the samples of the test group. We were able to detect 983 malware files using the malicious trees inferred from 193 malware files, and show that benign programs are benign, thus a 0% false positive rate.

### 6.1 Inferring malicious behaviors

To infer malicious behaviors, we transformed each of the 193 malware binary files into a **PDS** model using the approach described in Section 2. To implement the oracle  $\mathcal{O}$ , we use the *PoMMaDe* tool [25] that uses Jakstab [19] and IDA Pro [15]. Jakstab performs static analysis of the binary program. However, it does not allow to extract API functions information, so IDA Pro is used to obtain such information, thus obtaining  $\varrho_{ar}$  and  $\varrho_{\lambda}$ . The  $\varrho_{\tau}$  function was obtained by querying the available information in the MSDN website.

We apply Algorithm 1 to the **PDS** models to extract **SCDTs** for each of the malware instances. The current results were obtained with an  $h$  value of 2. In practice, to avoid the overapproximation of malicious trees, in the generation of **SCDTs** for the detection phase we consider the condition in line 6 of Algorithm 2,  $w_o[n] = w_d[m]$  true only when we know the value outputted by the oracle is precise.

To compute the **MalSCDT** we encode the extracted **SCDT** as graphs and try to calculate the most frequent subgraphs. We use the gSpan [30] tool for that, it computes frequent subgraph structures using a depth-first tree search over a canonical labeling of graph edges relying on the linear ordering property of the labeling to prune the search space. The tool has been applied in various domains as active chemical compound structure mining and its performance is competitive among other tools [29]. The tool supports only undirected graphs, therefore a mismatch with the trees (that can be seen as rooted, acyclic direct graphs) used in this work. The mismatch is overcome via a direction tag in the graph labels.

For the 193 files extracted **SCDTs** we have run the gSpan tool with support 0.6%. This is a tunable value for which we chose the one that allows better detection results. With this value we obtained 1026 subtrees (**MalSCDTs**), and best detection results. From the inferred malicious trees output from gSpan, we build a tree automaton recognizing such trees.

The training dataset contains 12 families of malware summarized in Table 1. In average, our tool extracts 7 **SCDTs** in 30 seconds for each malware file. To store the 1026 discovered **MalSCDTs** the automaton file used 24Kb of memory.

Name	#
Backdoor.Win32.Agent	26
Worm.Win32.AutoRun	13
Email-Worm.Win32.Bagle	19
Email-Worm.Win32.Batzback	4
Backdoor.Win32.Bifrose	46
Backdoor.Win32.Hupigon	5
Email-Worm.Win32.Kelino	7
Trojan-PSW.Win32.LdPinch	13
Email-Worm.Win32.Mydoom	26
Email-Worm.Win32.Nihilit	7
Backdoor.Win32.SdBot	14
Backdoor.Win32.Small	13
Total	193

**Table 1.** Training dataset

## 6.2 Detecting malware

Malware detection is reduced to generating **SCDTs** and checking whether they are recognized by the inferred automaton. Thus, to perform detection on an input binary file, we model it as **PDS** using the approach described in Section 2 and extract **SCDTs** using the approach detailed in Section 4.2. If any subtree of the extracted tree is recognized by the automaton recognizing the malicious behaviors, we decide the binary sample is malware. We implemented such procedure in our tool and were able to detect 983 malware samples from 330 different families.

In Table 2 we show the range of malware families and number of samples that our tool detects as malware. In average, our tool extracts 64 **SCDTs** in 2.15 seconds for each file (this value may be largely improved given that runtime efficiency was not a main goal of the prototype design). The discrepancy in the number of trees generated (compared to the training set) is justified by an implementation choice regarding the oracle approximation of the set of values pushed to the stack. In the generation of **SCDTs** for the detection phase we consider the condition in line 6 of Algorithm 2,  $w_o[n] = w_d[m]$  true even if the values are approximated. Such cases were discarded in the generation of **SCDTs** in the inference step where it holds only when the oracle outputs precise values. The automaton tree recognition execution time is negligible ( $< 0.08$  secs) in all cases. To check the robustness of the detector, we applied it to a set of 250 benign programs. Our tool was able to classify such programs as benign, obtaining a 0% false positive rate. In 88% of the cases the tool extracts **SCDTs** and at least in 44% of the files there is a call to a function involved in malicious behavior (e.g. *GetModuleFileName*, *ShellExecute*,...), but no tree was recognized as malicious. This value is in line with the values detailed in [10,14] and better than the 5% reported in [2].

## 7 Related work

Malicious behaviors have been defined in different ways. The foundational approaches via computable functions [1], based in Kleene’s recursion theorem [4,5,6], or the neat definition using MAlLog [20] capture the essence of such behaviors, but are too abstract to be used in practice or require the full specification of software functionality. Our work is close to the approaches using model checking and temporal logic formulas as malicious behavior specification [24,25]. In such works specifications have to be designed by hand while we are able to learn them automatically. Some of the trees we infer describe malicious behaviors encoded in such formulas.

Regarding semantic signature inference there are the works [10,14] where the extraction of behaviors is based on dynamic analysis of executables. From the execution traces collected, data flow dependencies among system calls are recovered by comparing parameters and type information. The outcome are dependence graphs where the nodes are labeled by system function names and the edges capture the dependencies between the system calls. Another dynamic analysis based approach is the one of [2] where trees, alike ours, express the same kind of data flows between nodes representing system calls. Both approaches are limited by the drawbacks of dynamic analysis. For instance, time limitations, limited system call tracing or an overhead up to  $90\times$



Name	#	Name	#	Name	#	Name	#
Backdoor.Win32.AF	1	Backdoor.Win32.MoonPie	1	Backdoor.Win32.UpRootKit	1	Email-Worm.Win32.Kelino	6
Backdoor.Win32.Afbot	1	Backdoor.Win32.Mowalker	1	Backdoor.Win32.Ursus	1	Email-Worm.Win32.Kergez	1
Backdoor.Win32.Afcore	6	Backdoor.Win32.Mtexer	2	Backdoor.Win32.Utilma	1	Email-Worm.Win32.Kipsis	2
Backdoor.Win32.Agent	66	Backdoor.Win32.Mydons	1	Backdoor.Win32.VB	2	Email-Worm.Win32.Kirbster	1
Backdoor.Win32.Agobot	47	Backdoor.Win32.Ncx	1	Backdoor.Win32.VHM	1	Email-Worm.Win32.Klez	9
Backdoor.Win32.Alcodor	1	Backdoor.Win32.NerTe	3	Backdoor.Win32.Vatos	1	Email-Worm.Win32.Lacrow	2
Backdoor.Win32.Antilam	9	Backdoor.Win32.NetControl	2	Backdoor.Win32.Verify	1	Email-Worm.Win32.Lara	1
Backdoor.Win32.Apdoor	6	Backdoor.Win32.NetShadow	1	Backdoor.Win32.WMFA	1	Email-Worm.Win32.Lentin	10
Backdoor.Win32.Assasin	3	Backdoor.Win32.NetSpy	8	Backdoor.Win32.WRT	1	Email-Worm.Win32.Locksky	2
Backdoor.Win32.Asylum	8	Backdoor.Win32.Netbus	2	Backdoor.Win32.WbeCheck	3	Email-Worm.Win32.Lohack	3
Backdoor.Win32.Avstral	2	Backdoor.Win32.Netdex	2	Backdoor.Win32.Webdor	6	Email-Worm.Win32.LovGate	3
Backdoor.Win32.BLA	2	Backdoor.Win32.Netpocalipse	1	Backdoor.Win32.Whisper	1	Email-Worm.Win32.Mescan	1
Backdoor.Win32.BNLite	1	Backdoor.Win32.Neurotic	2	Backdoor.Win32.Wilba	1	Email-Worm.Win32.Mimail	1
Backdoor.Win32.BO2K	6	Backdoor.Win32.Nuclear	3	Backdoor.Win32.Winker	5	Email-Worm.Win32.Miti	1
Backdoor.Win32.Bancodor	1	Backdoor.Win32.Nucledor	2	Backdoor.Win32.WinterLove	7	Email-Worm.Win32.Modnar	1
Backdoor.Win32.Bandok	1	Backdoor.Win32.Nyrobot	1	Backdoor.Win32.Wisdoor	7	Email-Worm.Win32.Mydoom	8
Backdoor.Win32.Banito	4	Backdoor.Win32.Optix	9	Backdoor.Win32.Wolff	4	Email-Worm.Win32.NWWF	1
Backdoor.Win32.Beastdoor	6	Backdoor.Win32.PPCore	1	Backdoor.Win32.XBot	1	Email-Worm.Win32.Navidad	1
Backdoor.Win32.Bifrose	5	Backdoor.Win32.PPDoor	2	Backdoor.Win32.XConsole	1	Email-Worm.Win32.NetSky	2
Backdoor.Win32.BoomRaster	1	Backdoor.Win32.Pacac	1	Backdoor.Win32.XLog	2	Email-Worm.Win32.NetSup	1
Backdoor.Win32.Breplibot	6	Backdoor.Win32.Padodor	5	Backdoor.Win32.Xdoor	2	Email-Worm.Win32.Netav	1
Backdoor.Win32.Bushtrommel	2	Backdoor.Win32.PcClient	12	Backdoor.Win32.Y2KCount	1	Email-Worm.Win32.Newapt	6
Backdoor.Win32.ByShell	1	Backdoor.Win32.PeepViewer	1	Backdoor.Win32.Ythac	1	Email-Worm.Win32.Nihilit	1
Backdoor.Win32.Cabrotor	1	Backdoor.Win32.Peers	2	Backdoor.Win32.Zerg	1	Email-Worm.Win32.Nirky	1
Backdoor.Win32.Cafeini	1	Backdoor.Win32.Penrox	1	Backdoor.Win32.Zombam	1	Email-Worm.Win32.Paroc	1
Backdoor.Win32.Cheng	1	Backdoor.Win32.Pepbot	1	Backdoor.Win32.Zomby	1	Email-Worm.Win32.Parrot	1
Backdoor.Win32.Cigivip	1	Backdoor.Win32.Pingdoor	1	Constructor.Win32.Delf	1	Email-Worm.Win32.Pepex	2
Backdoor.Win32.Cmjspy	8	Backdoor.Win32.Pipes	1	Constructor.Win32.ETVM	2	Email-Worm.Win32.Pikis	2
Backdoor.Win32.Cocozul	2	Backdoor.Win32.Plunix	1	Constructor.Win32.EvilTool	1	Email-Worm.Win32.Plage	1
Backdoor.Win32.Codbot	4	Backdoor.Win32.Pornu	1	Constructor.Win32.MS04-032	1	Email-Worm.Win32.Plexus	1
Backdoor.Win32.Coldfusion	3	Backdoor.Win32.Probot	1	Constructor.Win32.MS05-009	1	Email-Worm.Win32.Pnguin	1
Backdoor.Win32.Commlnet	3	Backdoor.Win32.Proxydor	2	Constructor.Win32.SP1	1	Email-Worm.Win32.Poo	1
Backdoor.Win32.Coredoor	1	Backdoor.Win32.Psychward	5	Constructor.Win32.SS	2	Email-Worm.Win32.Postman	1
Backdoor.Win32.Crunch	1	Backdoor.Win32.Prakks	1	Constructor.Win32.VCL	1	Email-Worm.Win32.Qizy	1
Backdoor.Win32.DKangel	2	Backdoor.Win32.Puddy	1	DoS.Win32.Asencode	1	Email-Worm.Win32.Rammer	1
Backdoor.Win32.DRA	4	Backdoor.Win32.R3C	1	DoS.Win32.Atker	1	Email-Worm.Win32.Rapita	1
Backdoor.Win32.DSNX	3	Backdoor.Win32.RAT	2	DoS.Win32.DStorm	1	Email-Worm.Win32.Rayman	1
Backdoor.Win32.DarkFip	3	Backdoor.Win32.RDR	1	DoS.Win32.Igempex	1	Email-Worm.Win32.Repah	2
Backdoor.Win32.DarkMoon	1	Backdoor.Win32.Rbot	8	DoS.Win32.SQLStorm	1	Email-Worm.Win32.Ronoper	20
Backdoor.Win32.Delf	31	Backdoor.Win32.Redkod	4	Email-Worm.Win32.Anar	2	Email-Worm.Win32.Roron	23
Backdoor.Win32.Dindang	1	Backdoor.Win32.Revenge	1	Email-Worm.Win32.Android	1	Email-Worm.Win32.Sabak	1
Backdoor.Win32.DragonIrc	1	Backdoor.Win32.Rirc	1	Email-Worm.Win32.Animan	1	Email-Worm.Win32.Savage	2
Backdoor.Win32.Dumador	3	Backdoor.Win32.Robobot	1	Email-Worm.Win32.Anpir	1	Email-Worm.Win32.Scaline	1
Backdoor.Win32.Expir	1	Backdoor.Win32.Ronater	1	Email-Worm.Win32.Ardurk	2	Email-Worm.Win32.Scrambler	1
Backdoor.Win32.HacDef	2	Backdoor.Win32.Rootcip	1	Email-Worm.Win32.Asid	1	Email-Worm.Win32.Seliz	1
Backdoor.Win32.Hackarmy	3	Backdoor.Win32.Roron	1	Email-Worm.Win32.Assarm	1	Email-Worm.Win32.Sharpei	1
Backdoor.Win32.Hupigon	4	Backdoor.Win32.RtKit	4	Email-Worm.Win32.Atak	1	Email-Worm.Win32.Silly	1
Backdoor.Win32.IRCBot	6	Backdoor.Win32.Ruleodor	4	Email-Worm.Win32.Avron	2	Email-Worm.Win32.Sircam	1
Backdoor.Win32.lerk	1	Backdoor.Win32.Sping	3	Email-Worm.Win32.Bagle	3	Email-Worm.Win32.Skudex	2
Backdoor.Win32.Jacktron	1	Backdoor.Win32.SatanCrew	1	Email-Worm.Win32.Bagz	5	Email-Worm.Win32.Sonic	4
Backdoor.Win32.Jeemp	1	Backdoor.Win32.Sbot	2	Email-Worm.Win32.Banof	1	Email-Worm.Win32.Stator	1
Backdoor.Win32.Katherdoor	7	Backdoor.Win32.SdBot	63	Email-Worm.Win32.Bater	1	Email-Worm.Win32.Stopin	3
Backdoor.Win32.Katien	2	Backdoor.Win32.Seed	3	Email-Worm.Win32.Batzback	3	Email-Worm.Win32.Sunder	1
Backdoor.Win32.Ketch	4	Backdoor.Win32.Serman	1	Email-Worm.Win32.Blebla	1	Email-Worm.Win32.Svoy	2
Backdoor.Win32.Kidterror	1	Backdoor.Win32.ShBot	1	Email-Worm.Win32.Bumdoc	2	Email-Worm.Win32.Swen	1
Backdoor.Win32.Konik	1	Backdoor.Win32.Shakdos	1	Email-Worm.Win32.Charch	1	Email-Worm.Win32.Tanatos	3
Backdoor.Win32.Krepper	2	Backdoor.Win32.Shox	1	Email-Worm.Win32.Cholera	1	Email-Worm.Win32.Taripox	2
Backdoor.Win32.Labrus	1	Backdoor.Win32.SilverFTP	1	Email-Worm.Win32.Coronex	3	Email-Worm.Win32.Totilix	1
Backdoor.Win32.LanFiltrator	2	Backdoor.Win32.Sinf	1	Email-Worm.Win32.Cult	1	Email-Worm.Win32.Trilissa	4
Backdoor.Win32.LanaFTP	1	Backdoor.Win32.Sinit	4	Email-Worm.Win32.Delf	4	Email-Worm.Win32.Trood	2
Backdoor.Win32.Laocon	1	Backdoor.Win32.SkyDance	1	Email-Worm.Win32.Desos	1	Email-Worm.Win32.Unis	1
Backdoor.Win32.Latinus	5	Backdoor.Win32.Small	22	Email-Worm.Win32.Donghe	3	Email-Worm.Win32.Urbe	3
Backdoor.Win32.Lemerul	1	Backdoor.Win32.Sporkbot	1	Email-Worm.Win32.Drefir	1	Email-Worm.Win32.Valha	1
Backdoor.Win32.Lesbot	1	Backdoor.Win32.SpyBoter	9	Email-Worm.Win32.Duksten	2	Email-Worm.Win32.Volag	1
Backdoor.Win32.Levelone	2	Backdoor.Win32.Stang	1	Email-Worm.Win32.Dumaru	10	Email-Worm.Win32.Vorgon	2
Backdoor.Win32.Liondoor	1	Backdoor.Win32.Stats	1	Email-Worm.Win32.Energy	1	Email-Worm.Win32.Warevov	1
Backdoor.Win32.Lithium	3	Backdoor.Win32.Stigmador	1	Email-Worm.Win32.Entangle	1	Email-Worm.Win32.Winevar	1
Backdoor.Win32.Litmus	1	Backdoor.Win32.SubSeven	1	Email-Worm.Win32.Epon	1	Email-Worm.Win32.Wozzer	1
Backdoor.Win32.LittleBusters	1	Backdoor.Win32.Sumatrix	1	Email-Worm.Win32.Eyeveg	3	Email-Worm.Win32.Xanax	2
Backdoor.Win32.LittleWitch	1	Backdoor.Win32.Suslix	1	Email-Worm.Win32.Fix2001	1	Email-Worm.Win32.Yanz	1
Backdoor.Win32.Livup	1	Backdoor.Win32.Symes	1	Email-Worm.Win32.Frethem	2	Email-Worm.Win32.Yenik	1
Backdoor.Win32.Lixy	1	Backdoor.Win32.Sysinst	1	Email-Worm.Win32.Frubece	1	Email-Worm.Win32.Zircon	4
Backdoor.Win32.Lurker	1	Backdoor.Win32.System33	1	Email-Worm.Win32.GOPworm	1	Exploit.Win32.Agent	3
Backdoor.Win32.Lyusane	1	Backdoor.Win32.Sytr	1	Email-Worm.Win32.Gift	2	Exploit.Win32.AntiRAR	1
Backdoor.Win32.MSNMaker	1	Backdoor.Win32.TDS	3	Email-Worm.Win32.Gismor	1	Exploit.Win32.CAN	1
Backdoor.Win32.MServ	1	Backdoor.Win32.Takit	1	Email-Worm.Win32.Gizer	2	Exploit.Win32.CVE-2006-1359	1
Backdoor.Win32.MainServer	1	Backdoor.Win32.Tasmer	1	Email-Worm.Win32.Gunsan	2	Exploit.Win32.CrobFTP	1
Backdoor.Win32.Matrix	3	Backdoor.Win32.Telemot	1	Email-Worm.Win32.Haltura	1	Exploit.Win32.DCom	3
Backdoor.Win32.Medbot	1	Backdoor.Win32.TheThing	3	Email-Worm.Win32.Hanged	1	Exploit.Win32.DameWare	1
Backdoor.Win32.Mellpon	2	Backdoor.Win32.Thunk	1	Email-Worm.Win32.Happy	1	Net-Worm.Win32.Muma	1
Backdoor.Win32.Metarage	1	Backdoor.Win32.Tonerok	3	Email-Worm.Win32.Ivalid	1	Trojan-PSW.Win32.LdPinch	16
Backdoor.Win32.Mhtserv	1	Backdoor.Win32.URCS	2	Email-Worm.Win32.Jeans	3	Worm.Win32.AutoRun	34
Backdoor.Win32.Micronet	1	Backdoor.Win32.Undernet	1	Email-Worm.Win32.Kadra	1		
Backdoor.Win32.MiniCommander	1	Backdoor.Win32.Unwind	1	Email-Worm.Win32.Keco	3		
						Total	983

Table 2. Test dataset name family and number of samples (#) detected

slower during execution [23]. Plus, from the dataset made publicly available in [2], we notice the signatures involve only functions from the Native API library. Our approach has the advantage of being API independent, thus the level of analysis may be tuned, plus Win32 API function based signatures should be shorter as each high level function should be translated into a set of calls to the Native API functions.

In [7] the authors propose to learn behaviors of binary files by extracting program control-flow graphs using dynamic analysis. Such graphs contain assembly instructions that correspond to control flow information e.g. *jmp*, but that introduces more possibilities to circumvent such signatures by rewriting the code. From the graphs, trees are computed and the union of all such trees is used to infer an automaton that is used in detection. Our inference does not output all the trees, only the most frequent, improving the learning process and generalizing from the training dataset.

An alternative to semantic signatures are works based on machine learning approaches as [28], which shows that by mining “*n*–grams” (a sequence of *n* bits), it is possible to distinguish malware from benign program. In our approach, the distinguishing features (malicious behaviors) can be seen as traces of program execution, thus having a meaning that can be more easily understood.

## 8 Conclusion

In this work, we have shown how to combine *static* reachability analysis techniques to infer malware semantic signatures in the form of malicious trees, which describe the data flows among system calls. Our experiments show that the approach can be used to automatically infer specifications of malicious behaviors and detect several malware samples from an a priori given smaller set of malware. We were able to detect 983 malware files using the malicious trees inferred from 193 malware files, and applied the detector to 250 benign files obtaining a 0% false positive rate.

As future work we envisage the improvement of the binary modeling techniques, for example enriching the function parameter type system to allow better approximations. The usage of more advanced mining techniques, e.g. structural leap mining used in [14], can be used to improve the learning approach. In another direction, given the relation between modal formulas and tree models a comparison between our approach and the approach in [24] concerning expressiveness and complexity is envisaged. Finally, a complexity study with respect to the depth of the trees extraction (parameter *h* in Algorithm 1) and size of the **HELTA** would be another alternative direction.

Summing up, the reachability analysis of **PDS** models of executables can play a major role in the malware specification inference domain. The ability to precisely analyze stack behavior enables the extraction of executables system call data flows and overcomes typical obfuscated calls to such routines.

## References

1. L. M. Adleman. An abstract theory of computer viruses. In *Proceedings of the 8th Annual Int. Cryptology Conference on Advances in Cryptology*, CRYPTO ’88, pages 354–374, 1988.

2. D. Babic, D. Reynaud, and D. Song. Malware analysis with tree automata inference. In *CAV*, pages 116–131, 2011.
3. J. Bergeron, M. Debbabi, M. M. Erhoui, and B. Ktari. Static analysis of binary code to isolate malicious behaviors. In *WETICE*, pages 184–189. IEEE Computer Society, 1999.
4. G. Bonfante, M. Kaczmarek, and J.-Y. Marion. *Toward an Abstract Computer Virology*. 2005.
5. G. Bonfante, M. Kaczmarek, and J.-Y. Marion. On Abstract Computer Virology from a Recursion Theoretic Perspective. *Journal in Computer Virology*, 1:45–54, 2006.
6. G. Bonfante, M. Kaczmarek, and J.-Y. Marion. *A Classification of Viruses Through Recursion Theorems*. 2007.
7. G. Bonfante, M. Kaczmarek, and J.-Y. Marion. Architecture of a morphological malware detector. *Journal in Computer Virology*, 5:263–270, 2009.
8. A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In *CONCUR*, pages 135–150, 1997.
9. M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns. In *Proceedings of the 12th conf. on USENIX Security Symposium*, 2003.
10. M. Christodorescu, S. Jha, and C. Kruegel. Mining specifications of malicious behavior. In *Proceedings of the 1st India software engineering conference, ISEC '08*, pages 5–14, 2008.
11. M. Christodorescu, S. Jha, S. A. Seshia, D. X. Song, and R. E. Bryant. Semantics-aware malware detection. In *IEEE Symposium on Security and Privacy*, pages 32–46, 2005.
12. J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *CAV*, pages 232–247, 2000.
13. M. Fossi, G. Egan, K. Haley, E. Johnson, T. Mack, T. Adams, J. Blackbird, M. Low, D. Mazurek, D. McKinney, et al. Symantec internet security threat report trends for 2010.
14. M. Fredrikson, S. Jha, M. Christodorescu, R. Sailer, and X. Yan. Synthesizing near-optimal malware specifications from suspicious behaviors. In *IEEE S. Security and Privacy*, 2010.
15. S. Hex-Rays. *Ida pro*, 2011.
16. A. Holzer, J. Kinder, and H. Veith. Using verification technology to specify and detect malware. In *EUROCAST*, pages 497–504, 2007.
17. J. Kinder, S. Katzenbeisser, C. Schallhart, and H. Veith. Detecting malicious code by model checking. In *DIMVA*, pages 174–187, 2005.
18. J. Kinder, S. Katzenbeisser, C. Schallhart, and H. Veith. Proactive Detection of Computer Worms Using Model Checking. *IEEE Trans. on Dependable and Secure Computing*, 2010.
19. J. Kinder and H. Veith. Jakstab: A static analysis platform for binaries. In *CAV*, 2008.
20. S. Kramer and J. C. Bradfield. A general definition of malware. *Journal in computer virology*, 6(2):105–114, 2010.
21. McAfee. McAfee threats report: Third quarter 2012. Technical report, McAfee, 2012.
22. P. Singh and A. Lakhotia. Static verification of worm and virus behavior in binary executables using model checking. In *Information Assurance Workshop*, pages 298–300, 2003.
23. A. Skaletsky, T. Devor, N. Chachmon, R. S. Cohn, K. M. Hazelwood, V. Vladimirov, and M. Bach. Dynamic program analysis of Microsoft Windows applications. In *ISPASS*, 2010.
24. F. Song and T. Touili. Efficient malware detection using model-checking. In *FM*, 2012.
25. F. Song and T. Touili. Pushdown model checking for malware detection. In *TACAS*, 2012.
26. F. Song and T. Touili. LTL model-checking for malware detection. In *TACAS*. 2013.
27. P. Szor. *The Art of Computer Virus Research and Defense*. Addison-Wesley Pro., 2005.
28. G. Tahan, L. Rokach, and Y. Shahar. Mal-id: Automatic malware detection using common segment analysis and meta-features. *Journal of Machine Learning Research*, 1:1–48, 2012.
29. M. Wörlein, T. Meinl, I. Fischer, and M. Philippsen. A quantitative comparison of the subgraph miners MoFa, gSpan, FFSM, and Gaston. *Knowledge Discovery in Databases*, 2005.
30. X. Yan and J. Han. gSpan: Graph-based substructure pattern mining. In *ICDM*, 2002.